Graph Indexing and Querying Using Fingerprint of Discriminative Frequent Fragments

Yu Wai Hlaing, Kyaw May Oo
University of Computer Studies, Yangon, Myanmar
yuwaihlaing.1987@gmail.com,kmayoo19@gmail.com

Abstract

Graphs are widely used to model complex structured data. Given a graph query, it is desirable to retrieve graphs quickly from a large database via graph-based indices. In this paper, we propose a graph indexing and querying model based on discriminative frequent fragments and fingerprint of these fragments. There are two main steps: index construction and query processing. In index construction phase, edge dictionary is used to simplify the process and to generate ids of graph edges. To reduce the size of index structure, two techniques, size-increasing support constraint and discriminative fragments are used. Canonical codes are then constructed based on discriminative frequent fragments. Then DB fingerprint is built based on codes. In query processing phase, query graph is parsed and canonical codes are constructed. Query's codes and DB fingerprint's codes are compared to get candidate answer set. Finally, candidate answer set is verified to ensure the query graph really contains in it or not by performing simple subgraph isomorphism test on each graph one by one.

1. Introduction

Recent technological and scientific advances have resulted in an abundance of data that describe and model phenomena in terms of primitive components and relationships between them. Graphs are used to model complex data objects in a number of applications, e.g., network intrusion detection [1, 2], the semantic web [3], behavioral modeling [4, 5], VLSI reverse engineering [6], Link analysis [7,8], and chemical compound classification [9, 10].

At the core of many graph-related applications, lies a common and critical problem: how to efficiently process graph queries and retrieve related graphs. In some cases, the success of an application directly relies on the efficiency of the query processing system. The classical graph query problem can be described as follows: Given a graph database $D = \{g_1, g_2, \ldots, g_n\}$ and a graph query q, find all the graphs in which q is a subgraph. It is inefficient to perform a sequential scan on the graph database and check whether q is a subgraph of g_i. Sequential scan is very costly because one has to not only access the whole graph database but also check subgraph isomorphism.

Clearly, it is necessary to build graph indices in order to help processing graph queries. In this paper, the processing of graph query can be divided into two major steps:

(1) **Index construction**, which is a preprocessing step, performed before real query processing. It is done by a data mining procedure, essentially evaluating, and selecting indexing features (i.e., frequent fragments) of graphs in a database. After selecting discriminative frequent fragments, canonical codes are constructed for these frequent fragments. And then DB fingerprint of the codes is built.

(2) **Query processing**, which consist of two steps: (i) Search, which parse the query graph and then canonical codes are constructed. Then, the codes of query and DB fingerprint are compared to get candidate answer set of query graph (ii) Verification, which verify the candidate answer set to ensure the query graph really contains in it or not by performing simple subgraph isomorphism test on each graph one by one.

The remainder of the paper is organized as follows: related work is presented in section 2. Section 3 defines the preliminary concepts, section 4 describes our indexing and querying mechanism, insert/delete maintenance is presented in section 5, section 6 identifies advantages and disadvantages of proposed technique and final conclusions are drawn in section 7.

2. Related Work

Several indexing techniques [17] have been developed for graph queries. Shasha et al. [11] proposed a path-based technique called GraphGrep. GraphGrep enumerates paths up to a threshold length from each graph. An index table is constructed where each row stands for a path and each column stands for a graph. Each entry in the table is the number of occurrences of the path in the graph. Queries are processed in two phases. The filtering phase generates a set of candidate graphs for which the count of each path is at least that of the query. The verification phase verifies each candidate graph by subgraph isomorphism and returns the answer set.

Yan et al. [12] proposed GIndex that uses frequent patterns as index features. Frequent patterns reduce the index space as well as improve the filtering rate. In a subsequent paper, the authors have extended their idea to partial matches of given queries [13].

He and Singh [14] develop a tree-based index called Closure-Tree, or C-tree, in which each node of the tree contains discriminative information about its descendants to facilitate effective pruning work. The summary information is like a "bounding box" of the structural information of the constituent graphs and it is represented as a graph closure. In C-tree indexing approach, the summary graph in the index structure is a generalized graph which is a structural union of the underlying database graphs. A pair wise graph comparison performed through heuristic techniques. For subgraph, the subgraph isomorphism problem is handled by an approximation technique called pseudo subgraph isomorphism. And for similarity queries, graph similarity is defined based on edit distance and then compute it using heuristic graph mapping methods.

Zhao et al [15] proposed a new tree-based indexing approach, which is called (Tree+Δ). The strategy is to first select frequent tree-features as the basis of a graph index, and then on-demand selects a small number of discriminative graph-features that can prune graphs better than the tree-features just selected, without performing costly graph mining in the beginning. This study proposed a new approach to estimate the pruning power of a graph-feature by its subtree-features with upper/lower bounds.

James Cheng et al [16] have worked on a new indexing method on graph databases based on frequent subgraphs by constructing a nested inverted index. This method is based on defining a set of frequent subgraphs that will define the index. This inverted graph index is composed of: -a Graph Array, which is the association of ID with the δ-Tolerance Closed Frequent subgraphs
in the set of graphs,
- an Edge Array, which stores the set of edges of the set of graphs,
- a list of ID-entries.

A graph g is a δ-Tolerance Closed Frequent subgraph if and only if $g \in F$ where F is the set of Frequent subgraphs. A graph g_1 is considered a Frequent subgraph if $fre(g_1) > (\sigma \cdot |Database|)$ where $freq(g_1) = |\{g_1', g_1' \in Database, g_1' \supseteq g_1\}|$ and $(0 < \sigma < 1)$ is a user-specified minimum frequency threshold. An ID-entry is a list of ID-arrays which are sets of IDs of the δ-Tolerance Closed Frequent subgraphs in the set of graphs according to the number of times they contain a specific edge.

3. Preliminary Concepts

In this section, we introduce the terminology used in this paper and give the formal problem definition. As an important data structure, the labeled graph is used to model complicated structures and schemaless data, e.g., XML is a kind of directed labeled graph, and chemical compounds are undirected labeled graphs. In this paper, we propose the graph indexing and querying method for undirected labeled graphs; however it is easy to extend our method to directed labeled graphs. Some definitions are presented as follows:

Definition 1 A **labeled graph** G is a five element tuple G = {V, E, ΣV, ΣE, l} where V is a set of vertices and E ⊆ V × V is a set of undirected edges. ΣV and ΣE are the sets of vertex labels and edge labels respectively. The labeling function l defines the mappings V → ΣV and E → ΣE.

Definition 2 A labeled graph G = (V, E, ΣV, ΣE, l) is **isomorphic** to another graph G′ = {V′, *E′, ΣV′, ΣE′, l′}, denoted by G ≈ G′, if there exists a bijection f : V → V′ such that*

1. ∀u ∈ V, (l (u) = l′ (f (u))),
2. ∀u, v ∈ V, ((u, v) ∈ E ⇔ (f (u), f (v)) ∈ E′), and
3. (u, v) ∈ E, (l(u, v) = l′ (f(u), f(v))).

The bijection f is called an isomorphism between G and G′. We also say that G is isomorphic to G′ and vice versa. A graph **automorphism** is an isomorphism from G to itself.

Definition 3 A labeled graph G is **subgraph isomorphic** to a labeled graph G′, denoted by G ⊆ G′, if there exists a subgraph G″ of G′ such that G is isomorphic to G″.

Definition 4 Given a graph database D = {g_1, g_2, ..., g_n}, the **support set** of graph g, denoted by D_g, is defined as the subset of D to which g is subgraph isomorphic. $D_g = \{g_i | g \subseteq g_i, g_i \in D\}$.

Problem Statement: In this paper, we investigate the following **graph query** problem. For a graph database *D* and a query graph *q*, find $D_q \in D$ that support *q*.

Figure 1 and 2 show sample graph database and sample query graph of chemical compounds respectively.

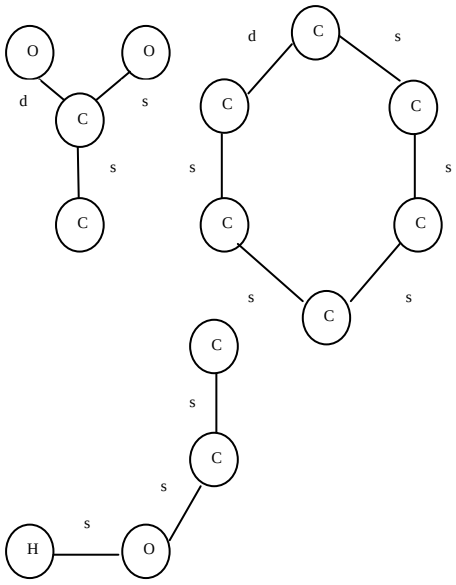

Figure 1. Sample Graph Database

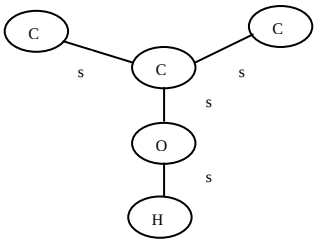

Figure 2. Sample Query Graph

4. Proposed Indexing Technique

The processing of graph queries in our technique can be divided into two major steps: index Construction and query processing.

4.1. Index Construction

Index construction is a preprocessing step which is performed before real query processing.

4.1.1. Edge Dictionary

The edge dictionary contains two parts: edge identifier (ID) and edge in the graph (Edge) which is the unique edge containing in the database. In the dictionary, an edge e is defined as 3-tuple (u, l_e, v) where u and v are labels of the vertices and l_e is the labels of the edge itself. Each edge appears only once in the edge dictionary, no matter how many times it appears in the graphs.

When a graph introduces in the graph database, the distinct edges are taken from the graph. And then it needs to check whether these edges already exist in the edge dictionary. If the dictionary contains these edges, we look up the corresponding identifier of that edge and use the identifier for further processing. If the edge is not in the dictionary, the edge is then inserted into the dictionary and the corresponding ID of the edge dictionary increases serially. Most graphs in

the chemical compound database have most similar edges and vertices.

Table 1. Example Edge Dictionary

ID	Edge
1	<H,s,O>
2	<O,d,C>
3	<O,s,C>
4	<O,s,H>
…	…

4.1.2. Frequent Fragments

Given a graph database D, $|D_g|$ is the number of graphs in D where g is a subgraph. $|D_g|$ is called (absolute) support, denoted by support(g). A graph g is frequent if its support is no less than a minimum support threshold, minSup. As one can see, frequent graph is a relative concept. Whether a graph is frequent or not depends on the setting of minSup. We use the term “fragment” to refer to a small subgraph (i.e., substructure) existing in graph databases and query graphs. In order to reduce the overall index size, it is appropriate for the index scheme to have low minimum support on small fragments (for completeness) and high minimum support on large fragments (for compactness). This criterion on the

selection of frequent fragments for effective indexing is called size-increasing support constraint.

Size-increasing Support:Given a monotonically nondecreasing function, f(l), pattern g is frequent under the size-increasing support constraint if and only if support(g) >= f(len(g)), and f(l) is a size-increasing support function. By enforcing the size-increasing support constraint, we bias to small fragments with low minimum support and large fragments with high minimum support. Especially, we always choose the (absolute) minSup to be 1 for size-0 fragment to ensure the completeness of the indexing. This method leads to two advantages: (1) the number of frequent fragments so obtained is much less than that with the lowest uniform minSup, and (2) low-support large fragments may be indexed well by their smaller subgraphs; thereby we do not miss useful fragments for indexing. The graph ids are associated with their corresponding frequent fragments.

By using frequent fragments with the size-increasing support constraint, we have a smaller number of fragments to index. However, the number of indexed fragments may still be huge when the support is low. For example, 1,000 graphs may easily produce 100,000 fragments of that kind. It is both time and space consuming to index them. Discriminative fragments are selected to get the best fragments, from the frequent fragments. In the end, only the most useful fragments are retained as indexing features.

4.1.3. Discriminative Frequent Fragments

If two similar frequent fragments, f_1 and f_2, are contained by the same set of graphs in the database, i.e., $D_{f1} = D_{f2}$, it is probably wise to include only one of them in the feature set. That is to say, if f_0, a supergraph of f, has the same support as f, it will not be able to provide more information than f if both are selected as indexing features. Thus f_0 should be removed from the feature set. In this case, we say f_0 is not more discriminative than f.

Redundant Fragment: Fragment x is redundant with respect to feature set F if

$$D_x \approx_{F \in f \wedge f \subseteq x} D_f \quad (1)$$

Where D_x is the set of graphs containing x and D_f is the set of graph containing frequent fragment f. In (1), each graph in set $\cap_{F \in f \wedge f \subseteq x} D_f$ contains all x's subgraphs in the feature set F. If D_x is close to $\cap_{F \in f \wedge f \subseteq x} D_f$, it implies the presence of fragment x in a graph can be predicted well by the presence of its subgraphs. Thus, fragment x should not be used as an indexing feature since it does not provide any benefit to pruning if its subgraphs are already being used as indexing features. In such case, x is a redundant fragment. In contrast, there are fragments which are not redundant, called discriminative fragments.

Discriminative Fragment: Fragment x is discriminative with respect to F if

$$D_x << _{F \in f \wedge f \subseteq x} D_f \quad (2)$$

In (2), since D_x is always a subset of $\cap_{F \in f \wedge f \subseteq x} D_f$, x should be either redundant or discriminative. Obviously, redundant fragment is a relative concept. We provide a simple measure on the degree of redundancy. Let fragments $f_1, f_2, \ldots, f_n$ be indexing fragments. We denote $1/Pr(x| f_1, f_2, \ldots, f_n)$ by □, called discriminative ratio. □ has the following properties:

1. □ >= 1.
2. when □ = 1, fragment x is completely redundant since the graphs indexed by this fragment can be fully indexed by the combination of fragment f_i' .
3. when □ >> 1, fragment x is more discriminative than the combination of fragments f_i' . Thus, x becomes a good candidate to index.
4. □ is related to the fragments which are already in the feature set.

The discriminative ratio can be calculated by the following formula:

$$□ = |\cap_i D_{fi}| / |D_x| \quad (3)$$

Where D_x is the set of graphs containing x and $\cap_i D_{fi}$ is the set of graphs which contain the fragments of f_i in the feature set.

4.1.4. Canonical Codes

Considering that graph isomorphism testing is hard; it is inefficient to scan the whole discriminative frequent fragment set to match with the query graph one by one. An efficient solution is to translate a graph into a sequence, called canonical label. If two fragments are the same, they must share the same canonical label.

Canonical codes are constructed by using edge ids of the discriminative frequent fragments. Frequent fragments are traversed in DFS traversal.

4.1.5. DB Fingerprint

After canonical codes of discriminative frequent fragments are constructed, DB fingerprint is built where each row stands for a canonical code and GraphID column stands for graph ids that contain discriminative fragments of this code.

Table 2. Example DB Fingerprint

Key	GraphID
1	$g_1,g_2,g_3,...$
2	$g_2,g_4,...$
23	$g_1,g_2,\ldots$
134	g_4
…	…

4.2. Query Processing

There are two steps in query processing phase. They are search and verification.

4.2.1. Search

In this step, query graph is parsed and construct canonical codes of its fragments. Then DB fingerprint is checked whether query canonical codes are contained in it or not. If the DB fingerprint code and query code are matched, the graphs of this code are included in candidate graph set. The process is continued until all of the query codes finished matching with DB fingerprint codes.

4.2.2. Verification

In verification, candidate answer set of the query graph is verified to ensure that the query graph really contains in it or not by performing simple subgraph isomorphism on each graph one by one. GraphGrep [11] proposed an alternative approach.

It records all the embeddings of paths in the graph database. It performs join operations on these embeddings to figure out the possible isomorphism mapping between the query graph and the graphs in candidate answer set. Considering there are lots of paths in the fingerprint and each path may have tens of embeddings, we find that in some cases it even performs worse than the simplest approach. Thus, we only implement the simple one in our study. The overall structure of our graph indexing mechanism is as follow:

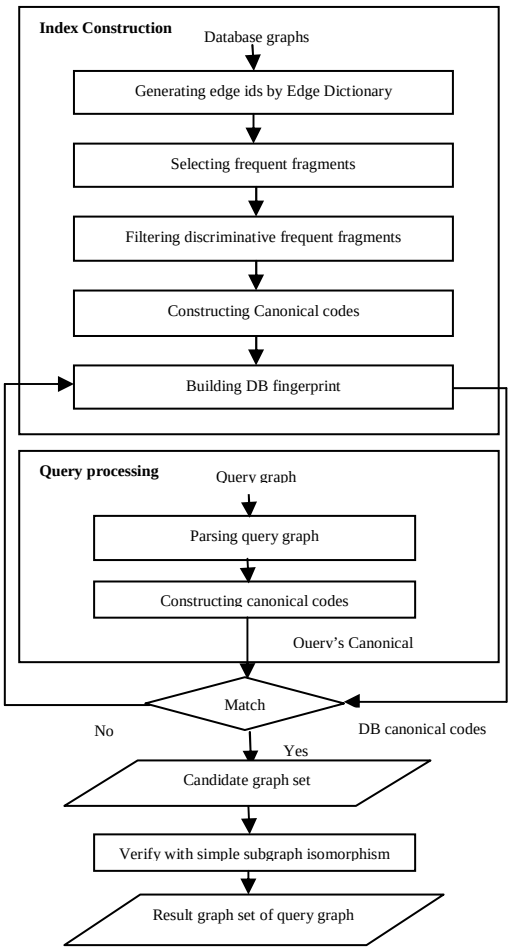

Figure 3. Overall Structure of Proposed Technique

5. Insert/Delete Maintenance

In the previous sections, we mostly focus on static indexing, i.e., we did not pay too much attention to updates. Our proposed method can be easily extended to the dynamic indexing. When a new graph g is inserted to the database, if the edges of g are already in edge dictionary we simply add g's id to the graph ids list of its corresponding codes in DB fingerprint. If the edge is new edge, it is inserted to the edge dictionary and follows the process. When a graph g is deleted from the database, we simply delete id of g from DB fingerprint and if g's id is occurred alone for some canonical codes, this row is deleted from DB fingerprint. This operation is efficient as long as the graphs in the database are homologous, i.e. representing similar objects, DB fingerprint will not change dramatically when the graph database varies.

However, in the case when there are too many insert/delete operations having performed on the database, e.g., one quarter of graphs in the database are changed, we can rebuild the DB fingerprint entirely to ensure the quality of the index.

6. Advantages and Disadvantages of Proposed Technique

In this section, we present an analysis of the efficiency of proposed technique. We identify the advantages and also discuss the disadvantages of this technique.

6.1. Advantages

Because of using frequent fragments as indexing feature instead of simple paths, we can avoid the problem of structural information lost and can reduce index construction time. By using the discriminative frequent fragments and size-increasing support, the size of index can be greatly reduced so the search space can be narrowed. In matching database graph and query, we use canonical codes of discriminative frequent fragments so index probing time is reduced.

6.2. Disadvantages

After candidate answer set is get, we need to verify that query graph is subgraph of candidate graphs. In this step we need to do expensive subgraph isomorphism test. So, candidate verification time is increased. It can reduce query performance.

7. Conclusion

In this paper, we propose an indexing and querying technique to efficiently retrieve the graph set that are supergraphs of query graph. Edge dictionary is used to generate ids of graph edges and to simply the matching process. Two techniques: the discriminative frequent fragments and size-increasing support are used in selecting indexing features; so the size of index can be greatly reduced. Canonical codes are used to make graph matching more easily. Verify the candidate answer set with simple subgraph isomorphism test to ensure that it really contain supergraphs of query graph or not.

References

[1] W. Lee and S. Stolfo, A framework for constructing features and models for intrusion detection systems, ACM transactions on information and system security, 2000.

[2] C. Ko, Logic induction of valid behavior specifications for intrusion detection, In IEEE Symposium on Security and Privacy, 2000.

[3] B. Berendt, A.Hotho, and G.Stumme, Towards semantic web mining. Proc. Of ISWC, 2002.

[4] S. Wasserman, K. Faust, and D. Iacobucci, Social network analysis : Methods and applications, Cambridge University Press, 1994.

[5] R. Mooney, P.Melville, L. Tang, J. Shavlik, I. Castro Dutra, and D. Page, Relational data mining with inductive logic programming for link discovery, AAAI Press/ The MIT Press, 2004.

[6] K. Yoshida and H. Motoda, CLIP: Concept learning from inference patterns, Artificial Intelligence, 1995.

[7] D. Jensen and H. Goldberg, Artificial Intelligence and Link Analysis, AAAI Press, 1998.

[8] C. Palmer, P. Gibbons, and C. Faloutsos, ANF: A fast and scalable tool for data mining in massive graphs, Proc. Of KDD, 2002.

[9] L. Dehaspe, H. Toivonen, and R. King, Finding frequent substructures in chemical compounds, Proc. Of KDD, 1998.

[10] M. Deshpande, M. Kuramochi, G. Karypis, Frequent sub-structure based approaches for classifying chemical compounds, Proc. Of ICDE, 2003.

[11] D. Shasha, J>T.L. Wang and R. Giugno, Algorithms and applications of tree and graph searching, 2002.

[12] X. Yan, P. S. Yu and J. Han, Graph Indexing: A Frequent Structures based approach, In SIGMOD Conference, 2004.

[13] X. Yan, P. S. Yu, and J. Han, Substructure similarity search in graph databases, In SIGMOD Conference, 2005.

[14] H. He and A. K. Singh, Closure-Tree: An Index Structure for Graph Queries, Proceedings of the 22nd International Conference on Data Engineering, 2006.

[15] P. Zhao, J. Yu and P. Yu, Graph indexing: Tree + Delta >= Graph. In Proceedings of the 33rd international conference on very large databases, 2007.

[16] J. Cheng, Y. Ke, W. Ng, and A. Lu, FG-Index: Towards Verification-Free Query Processing on Graph Databases, In SIGMOD '07, June 12-14, 2007.

[17] D. Dong and F. Liu, "Graph Database Indexing", University of Illinois Urbana Champaign, 2011.

[18] A. N. Thaing and K. M. Oo, "DAGC: Identification and Filtration of Automorphism Graphs in Graph Databases", International Journal of Computer Application (0975 - 8887), 2011.